

# Software Engineering Fundamentals: MS.NET

Method Calling

>  
**accenture**

Technology Solutions



# Contents

- What is a Method?
- Declaring a Method
- Method Calling
- Method Types
- Method Overloading
- Key Points

# Objectives

- At the end of this module, you should be able to:
  - Define a method
  - Demonstrate how to correctly declare a method
  - Demonstrate how methods call each other
  - Demonstrate parameter passing by value
  - Demonstrate parameter passing by reference
  - Demonstrate output parameters
  - Demonstrate parameter arrays (variable argument list)

# What is a Method?

- A **method** is a portion of code, referring to behaviors associated either with an object or its class.
  - It is used to **access and process** data contained in the object.
  - It is also used to provide responses to any messages received from **other objects**.
  - It is the **executable code** that **implements** the logic of a particular message for a class.
  - It is an operation or function that is associated with an object and is **allowed to manipulate** that **object's data**.

# Declaring a Method

- The following mentioned below is a general form of Method declaration.
  - Syntax

```
modifiers type Method-name (formal-parameter-list)
{
    method_body
}
```

- Method declaration consists of five components. Namely,
  - Methods Modifiers (modifiers)
  - Type of value the Method returns (type)
  - Name of the Method (method-name)
  - List of parameters (formal-parameter-list)
  - Body of the Method (method\_body)

# Declaring a Method (cont.)

- Steps in declaring a method:

1. Set the return type
2. Provide method name
3. Declare formal parameters

- Method signature

- consists of the method name and its parameters
- must be unique for each method in a class

- return statement

- allows the method to return a value to its caller
- also means to stop the execution of the current method and return to its caller
- implicit return at the end of the method

- A method that does not return a value must specify `void` as its return type

- A method with empty parameters

```
using System;

class Number
{
    int Multiply(int i, int j)
    {
        return i * j;
    }

    int Divide(int i, int j)
    {
        return i / j;
    }

    void PrintSum(int i, int j)
    {
        Console.WriteLine(i + j);
    }

    double Phi()
    {
        return 355.0 / 113.0;
    }
}
```

# Method Calling

```
using System;
public class CSharpMain
{
    public static void Main(String[] args)
    {
        // create a Person object
        Person you = new Person();
        you.talk();
        you.jump(3);
        Console.WriteLine(you.tellAge());

        // static keyword qualifies the method
        CSharpMain.talkOnly(you);

        // create object of main program
        CSharpMain me = new CSharpMain();
        me.jumpOnly(you);
    }
    static void TalkOnly(Person p) //static method
    {
        p.Talk();
    }
    void JumpOnly(Person p) // method
    {
        p.Jump(2);
    }
}
```

```
class Person
{
    public void Talk()
    {
        Console.WriteLine("blah, blah...");
    }
    public void Jump(int times)
    {
        for (int i = 0; i < times; i++)
        {
            Console.WriteLine("whoop!");
        }
    }
    public string TellAge()
    {
        return "I'm " + GetAge();
    }
    public int GetAge()
    {
        return 10;
    }
}
```

```
blah, blah...
whoop!
whoop!
whoop!
I'm 10
blah, blah...
whoop!
whoop!
```

# Method Types

- C# employs four kinds of parameters that are passed to methods.
  - Value Type parameters
    - Used for passing parameters into methods by **value**
  - Reference Type parameters
    - Used to pass parameters into methods by **reference**
  - Output parameters
    - Used to **pass results back** from a method
  - Parameter arrays
    - Used in a method definition to enable it **to receive variable number of arguments** when called

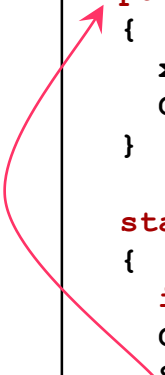


# Passing Value Type Parameters

- A value-type variable contains its data directly
  - By passing a value-type variable to a method it passes a copy of the variable to the method.
  - Changing the parameter value inside the method does not change the original data stored in the variable.

```
using System;
public class PassingValByValDemo
{
    public static void SquareIt(int x) // The parameter x is passed by value.
    {
        x *= x; //Changes to x will not affect the original value of x.
        Console.WriteLine("The value inside the method: {0}", x);
    }

    static void Main()
    {
        int n = 5;
        Console.WriteLine("The Value before calling the method: {0}", n);
        SquareIt(n); // Passing the variable by value.
        Console.WriteLine("The value after calling the method: {0}", n);
    }
}
```



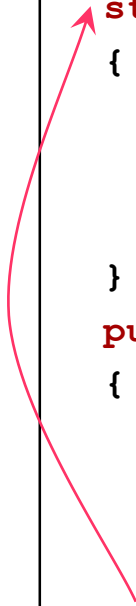
```
The value before calling the method: 5
The value inside the method: 25
The value after calling the method: 5
```

# Reference Parameters

- A variable of a reference type does not contain its data directly.
- It contains a reference to its data.
- By passing a reference-type parameter by reference, it is possible to change the data pointed to, such as the value of a class member.

```
i = 2 j = 3  
i = 3 j = 2
```

```
using System;  
class PassingValByRefDemo  
{  
    static void SwapByRef(ref int x, ref int y)  
    {  
        int temp = x;  
        x = y;  
        y = temp;  
    }  
    public static void Main()  
    {  
        int i = 2;  
        int j = 3;  
        Console.WriteLine($"i = {i} j = {j}");  
        SwapByRef(ref i, ref j);  
        Console.WriteLine($"i = {i} j = {j}");  
    }  
}
```



# Output Parameters

- `out` keyword causes arguments to be passed by reference
  - It is similar to the `ref` keyword
  - The difference is `ref` requires the variable to be initialized before being passed
- To use an `out` parameter, both the method definition and the calling method must explicitly use the `out` keyword

```
using System;
class OutReturnDemo
{
    static void Method(out int i, out string s1, out string s2)
    {
        i = 44;
        s1 = "I've been returned";
        s2 = null;
    }
    static void Main()
    {
        int value;
        string str1;
        string str2;
        Method(out value, out str1, out str2);
        // value is now 44
        // str1 is now "I've been returned"
        // str2 is (still) null;
        Console.WriteLine($"{value},{str1},{str2}");
    }
}
```

44,I've been returned,''

# Parameter Arrays

- The `params` keyword allows you to specify a *method parameter* that takes an *argument* where the *number of arguments is variable*.
- *No additional parameters* are permitted **after** the `params` keyword in a method declaration, and *only one* `params` *keyword* is permitted in a method declaration.

# Parameter Arrays (cont.)

```
using System;

public class MyClass
{
    public static void UseParams1
        (params int[] list)
    {
        foreach (var e in list)
        {
            Console.WriteLine(e);
        }
        Console.WriteLine();
    }

    public static void UseParams2
        (params object[] list)
    {
        foreach (var e in list)
        {
            Console.WriteLine(e);
        }
        Console.WriteLine();
    }
}
```

```
static void Main()
{
    UseParams1(1, 2, 3);
    // An array of objects can also be
    // passed, as long as
    UseParams2(1, 'a', "test");
    // the array type matches the method
    // being called.
    int[] myarray = new int[]
        { 10, 11, 12, 13 };
    UseParams1(myarray);
}
```

```
1
2
3

1
a
test

10
11
12
13
```

# Extension Methods

- Starting from C# 3.0 (and .NET 3.5) it is possible to define a method for a type that can be used as it would be type's own method:

```
public class Product
{
    public decimal Price { get; set; }
}

public static class Extensions
{
    public static decimal Discount(this Product product)
    {
        return product.Price * 0.9M;
    }
}

public class ExtensionsDemo
{
    public static void Main(string[] args)
    {
        var apple = new Product();
        apple.Price = 1.10M;
        Console.WriteLine(apple.Discount());
    }
}
```

# Named and Optional Parameters

- Starting from C# 4 it is possible to provide default values for method parameters:

```
void Method1(int required,  
             string optionalStr = "default value",  
             int optionalInt = 10) { ... }
```

- Valid method invocations:

```
Method1(10); // Method1(10, "default value", 10);  
Method1(14, "some value"); // Method1(14, "some value", 10);  
Method1(11, "another value", 15);  
Method1(11, optionalInt: 15);  
Method1(11, optionalInt: 15, optionalStr: "my value");  
Method1(optionalInt: 15, required: 10);
```

- Invalid method invocations:

```
Method1();  
Method1(11, , 15);
```

# Expression-Bodied Members

- Starting from C# 6.0 it is possible to declare single line methods.
- However, multi-line/statement lambdas are NOT allowed.

```
namespace Test
{
    using static System.Math;

    class Point
    {
        public double X { get; set; }
        public double Y { get; set; }

        public double Distance => Sqrt(X * X + Y * Y);

        public override string ToString() => $"({X}, {Y})";
    }
}
```



# Key Points

- Methods give objects their behavioral characteristics.
- A method must have a return type, a name, and optional parameters.
- The method signature refers to a method name and its parameters.
- Return statement returns a value to its caller or returns control to its caller.
- A method that does not return a value must specify void as a return type.
- Calls to a method should match its method signature.
- When calling a method in the same class, use only the method name (Nested Methods).
- When calling a method outside class, use the object reference.
- When calling a static method, use the class name.
- In C# method parameters are passed using pass by value, pass by reference, output parameters and parameter arrays.

# Questions and Comments

- What questions or comments do you have?

